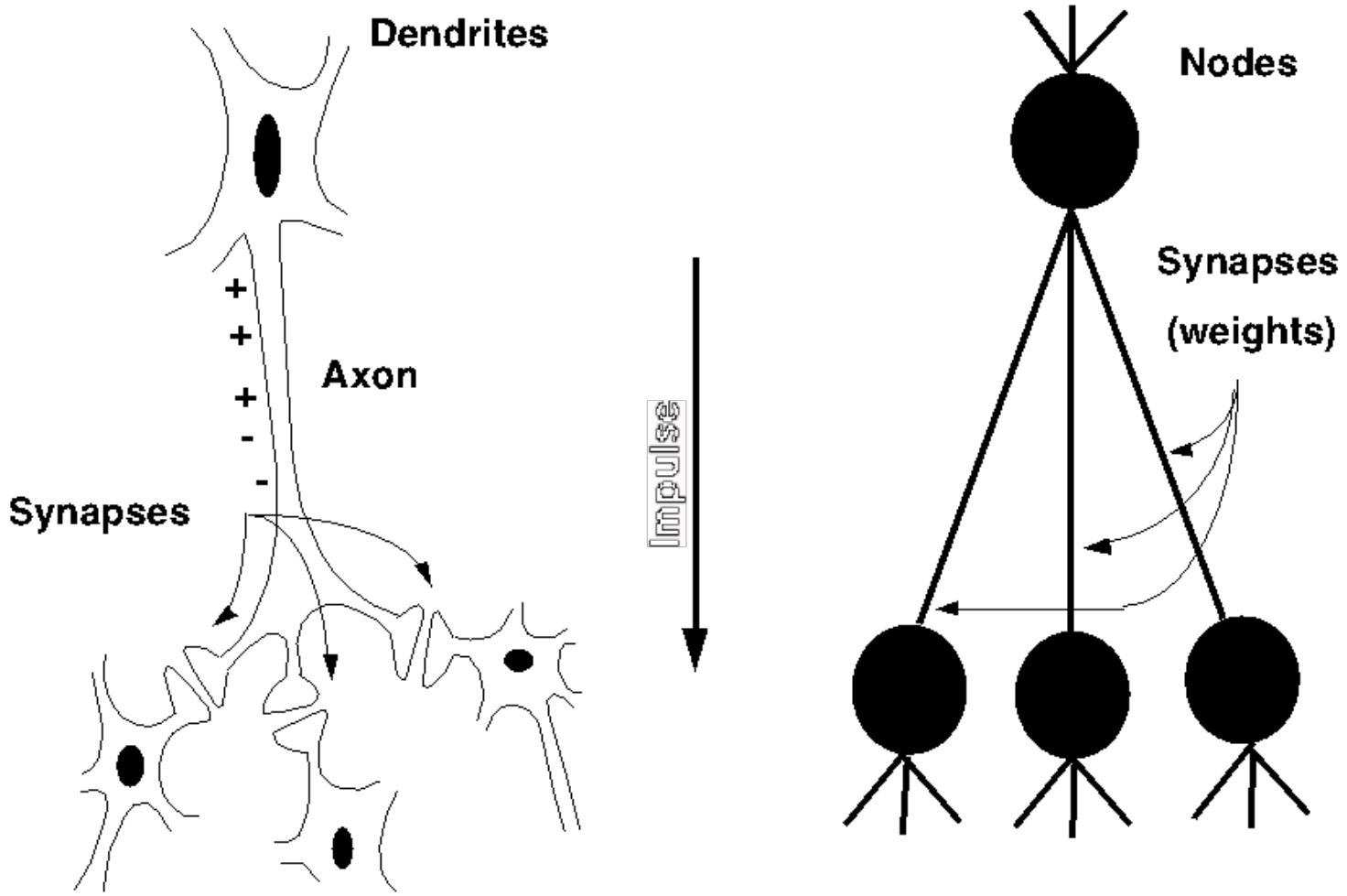
Machine Learning

Neural Networks

Biological Analogy

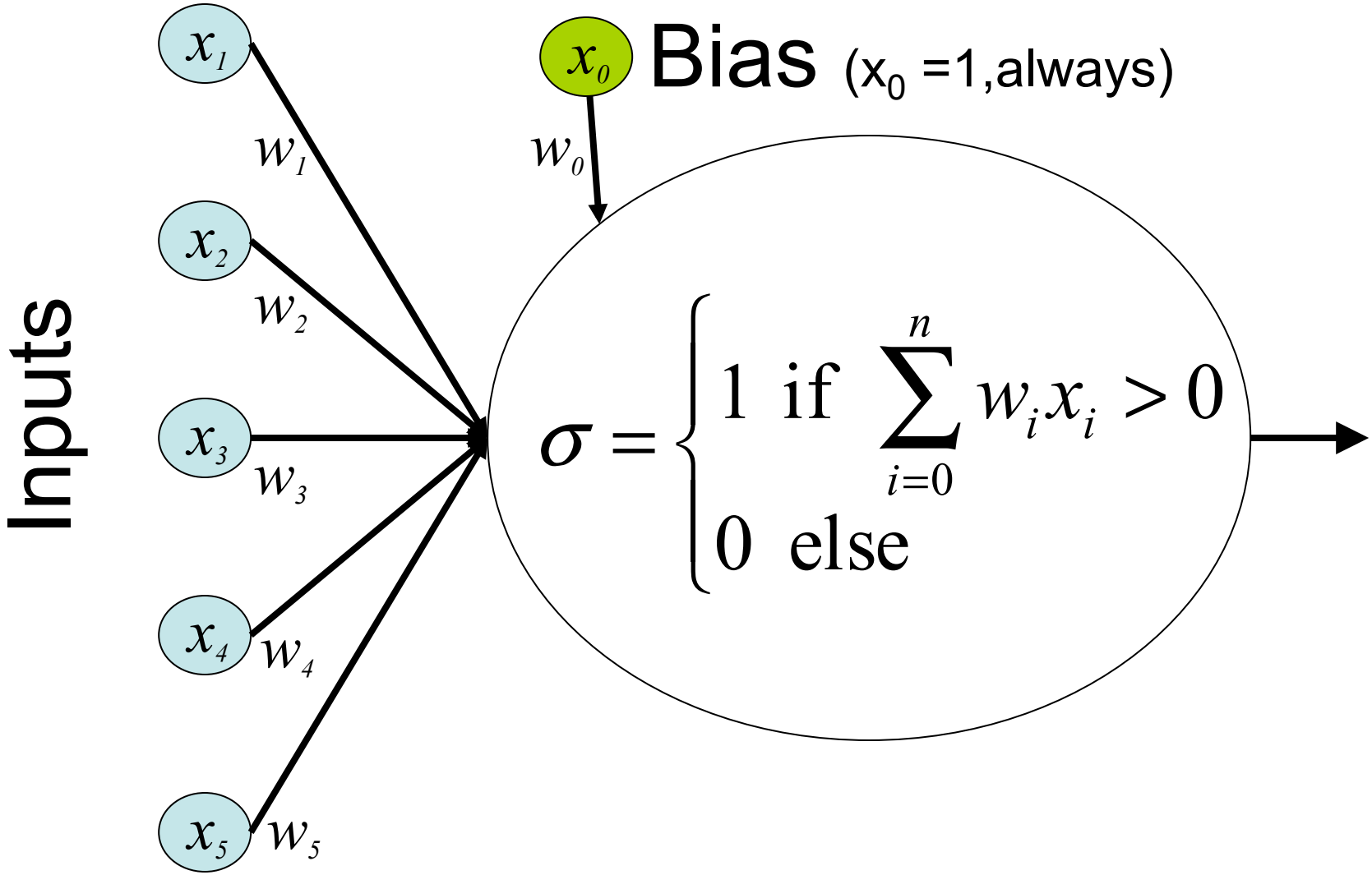


THE PERCEPTRON

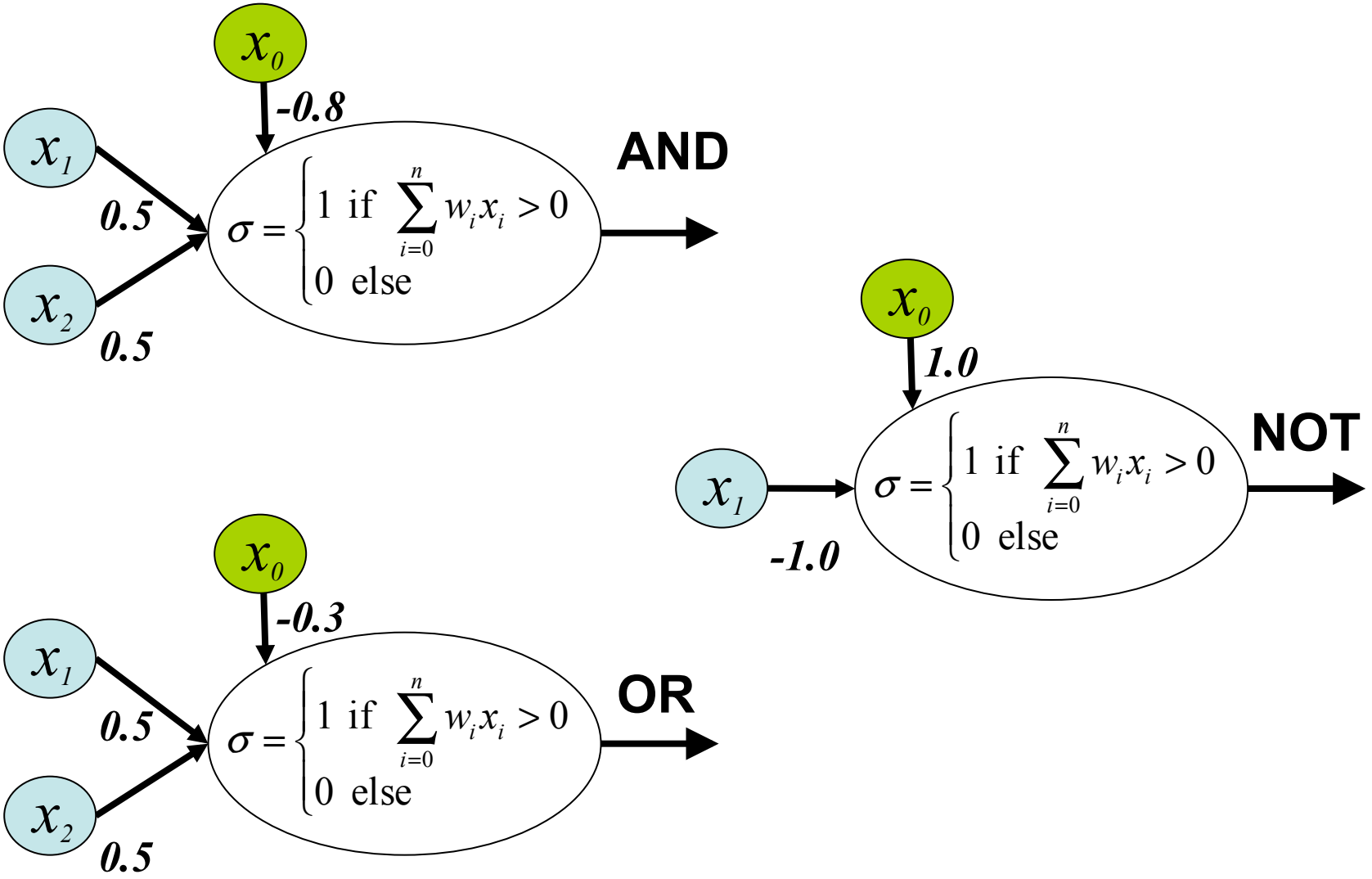
Perceptrons

- The "first wave" in neural networks
- Big in the 1960's
- Marvin Minsky and Seymour Papert
  - Perceptrons (1969)

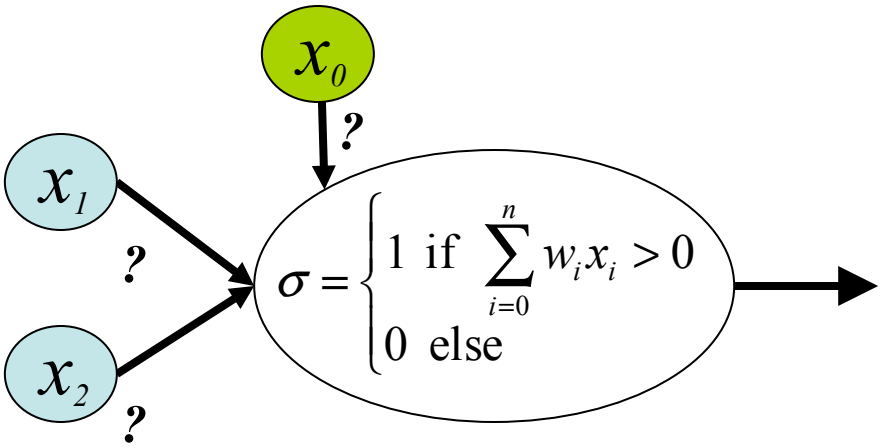
A single perceptron



Logical Operators

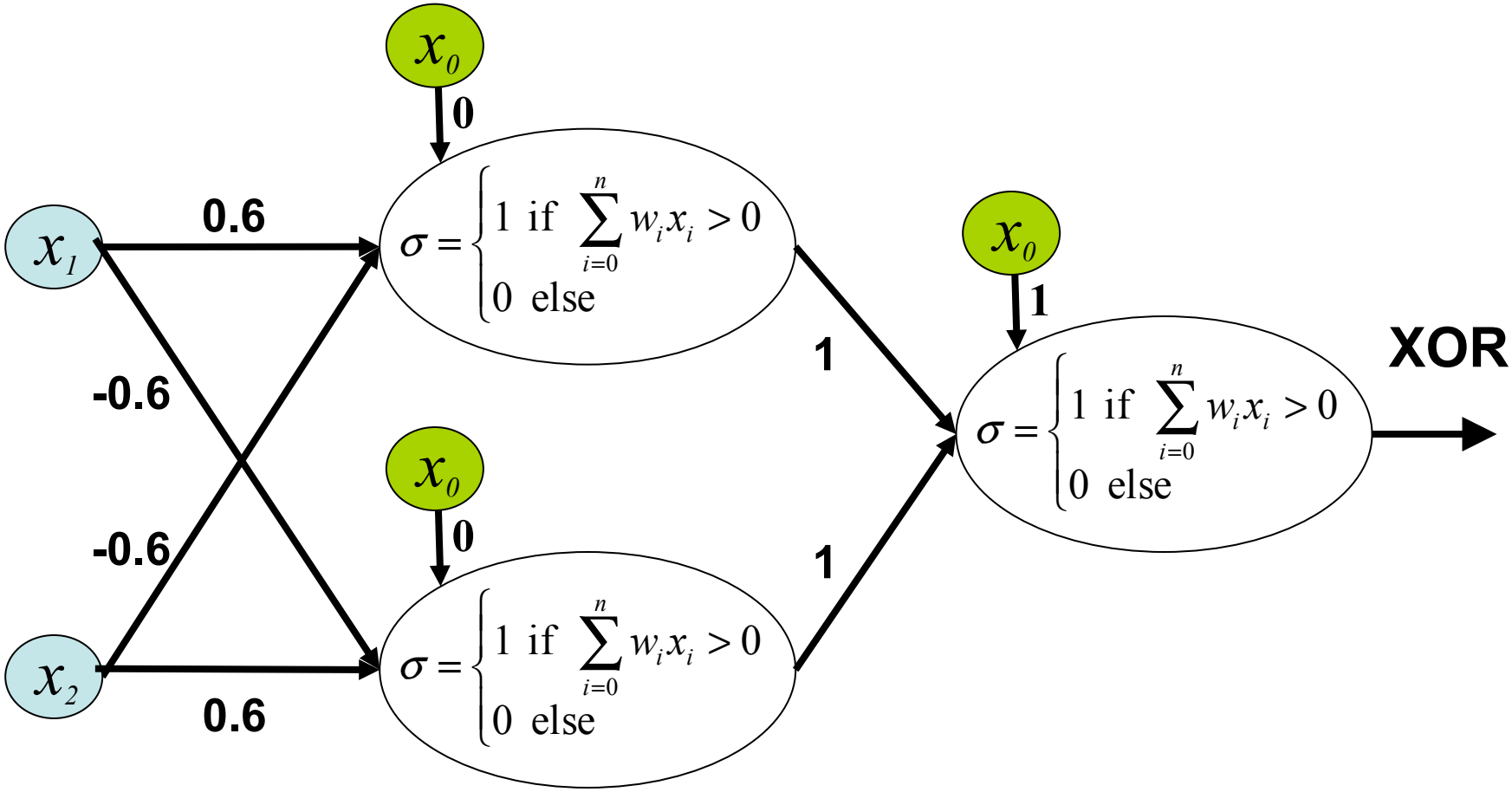


What weights make XOR?



- No combination of weights works
- Linear combinations don't allow this.
- No problem! Just combine perceptrons!

XOR

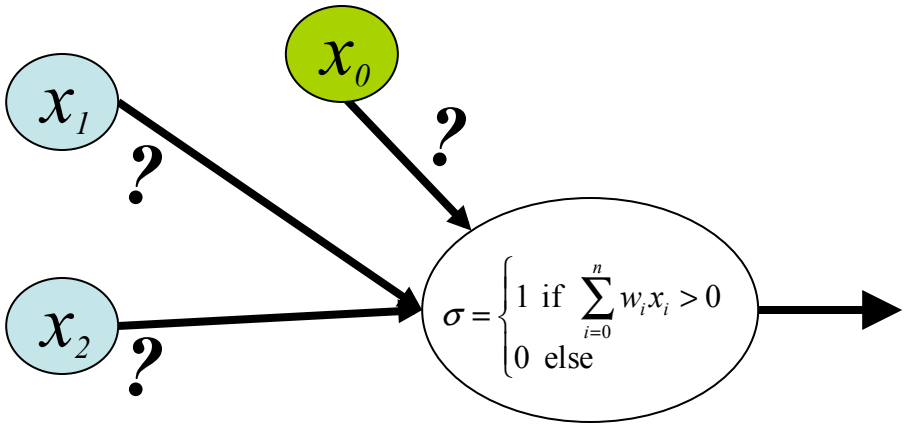

x_0
0
x_1
0.6
-0.6
x_0
0
-0.6
x_2
0.6
$\sigma = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ 0 \text{ else} \end{cases}$
1
$\sigma = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ 0 \text{ else} \end{cases}$
1
x_0
1
$\sigma = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ 0 \text{ else} \end{cases}$
XOR

The good news (so far)

- We can make AND, OR and NOT
- Combining those let us make arbitrary Boolean functions
- So…we're good, right?
- Let's talk about learning weights…

Learning Weights

- Perceptron Training Rule
- Gradient Descent
- (other approaches: Genetic Algorithms)



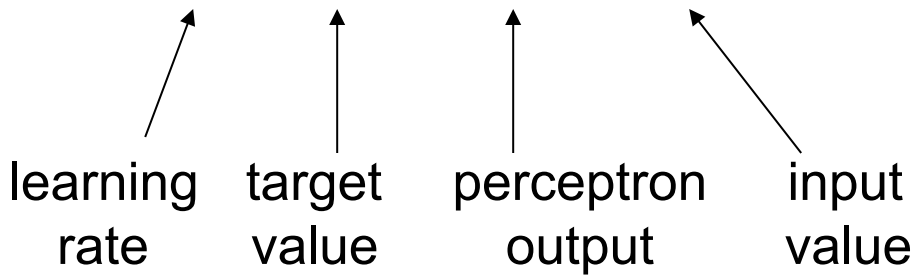
Perceptron Training Rule

- Weights modified for each example
- Update Rule:

$$w_i \leftarrow w_i + \Delta w_i$$

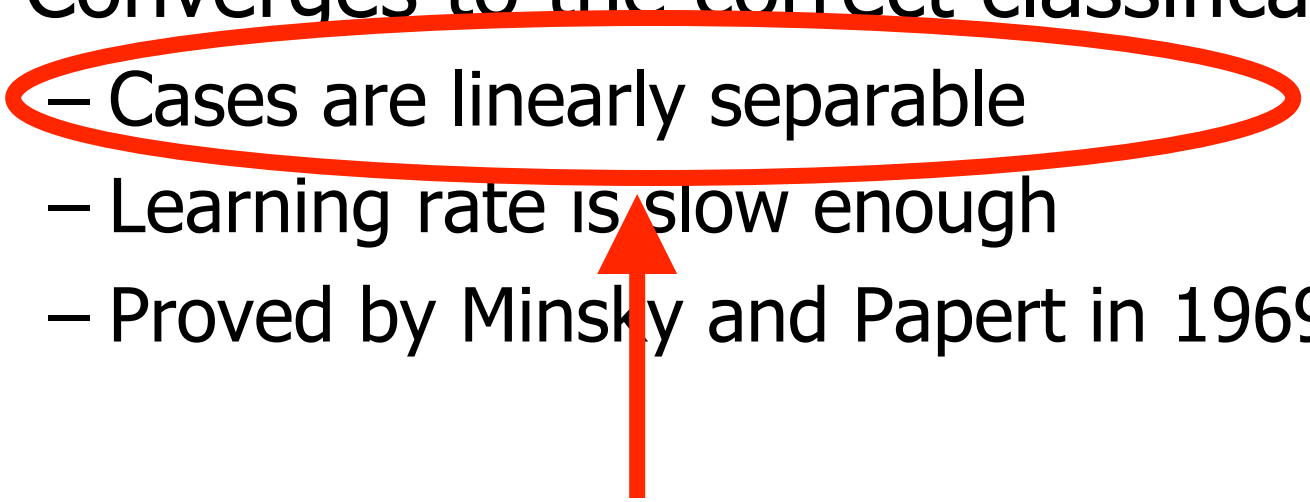
where

$$\Delta w_i = \eta (t - o) x_i$$



Perceptron Training Rule

- Converges to the correct classification IF
  - Cases are linearly separable
  - Learning rate is slow enough
  - Proved by Minsky and Papert in 1969



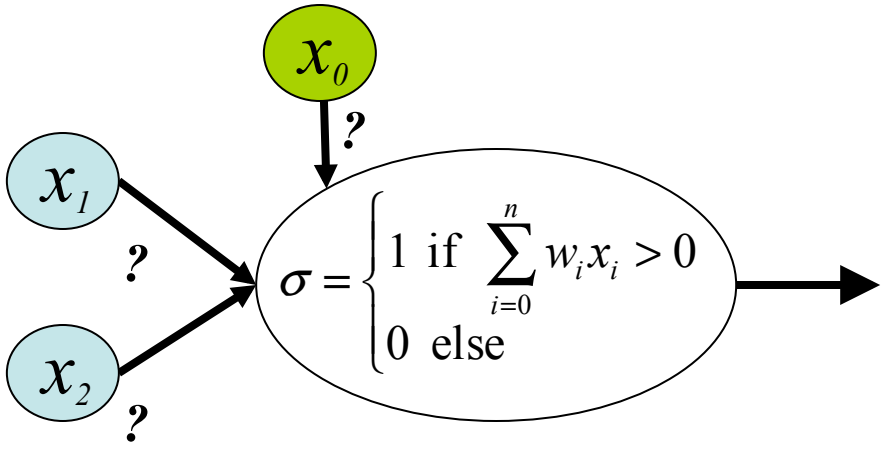
Killed widespread interest in perceptrons till the 80's

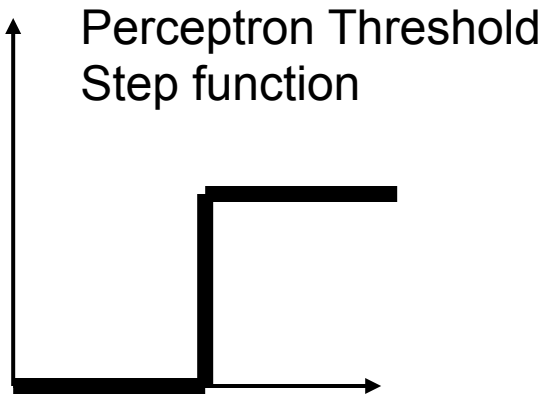
WHY?

What's wrong with perceptrons?

- You can always plug multiple perceptrons together to calculate any function.
- BUT…who decides what the weights are?
  - Assignment of error to parental inputs becomes a problem….
  - This is because of the threshold….
    - Who contributed the error?

Problem: Assignment of error



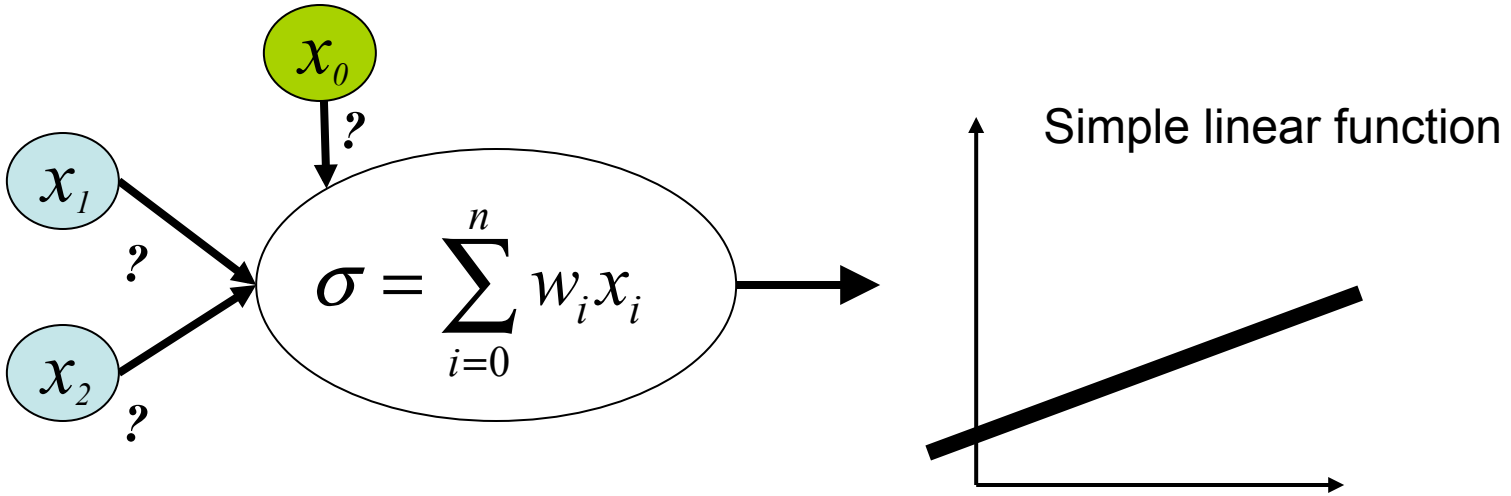


- Hard to tell from the output who contributed what
- Stymies multi-layer weight learning

LINEAR UNITS & DELTA RULE

Bryan Pardo, Northwestern University, Machine Learning EECS 349 Fall 2007

Solution: Differentiable Function



- Varying any input a little creates a perceptible change in the output
- This lets us propagate error to prior nodes

Measuring error for linear units

- Output Function

$$o(\vec{x}) = \vec{w} \cdot \vec{x}$$

- Error Measure:

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

data

target value

linear unit output

Gradient Decsent Rule

- To figure out how to change each weight to best reduce error we take the derivative w.r.t that weight

$$\frac{\partial E}{\partial w_i} \equiv \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

- …and then do some math-i-fication

The "delta" rule for weight updates

....YADA YADA YADA (this is in the book)

$$\frac{\partial E}{\partial w_i} \equiv \sum_{d \in D} (t_d - o_d)(-x_{i,d})$$

- ...and this gives our weight update rule

$$w_i \leftarrow w_i + \eta \sum_{d \in D} (t_d - o_d) x_{i,d}$$

Learning Rate

Input on example

Delta rule vs. Perceptron rule

- Perceptron Rule & Threshold Units
  - Learner converges on an answer ONLY IF data is linearly separable
  - Can't assign proper error to parent nodes

- Delta Rule & Linear Units
  - Minimizes error even if examples are not linearly separable
  - Linear units only make linear decision surfaces (can't make XOR even with many units)

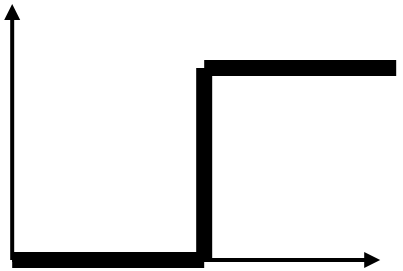
THE MULTILAYER PERCEPTRON

Bryan Pardo, Northwestern University, Machine Learning EECS 349 Fall 2007

A compromise function

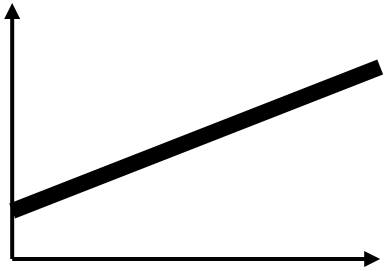
- Perceptron

$$output = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ 0 \text{ else} \end{cases}$$



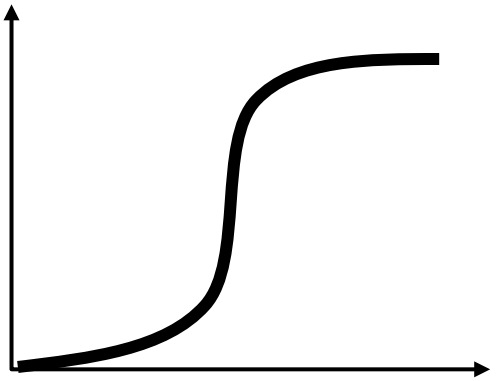
- Linear

$$output = net = \sum_{i=0}^{n} w_i x_i$$



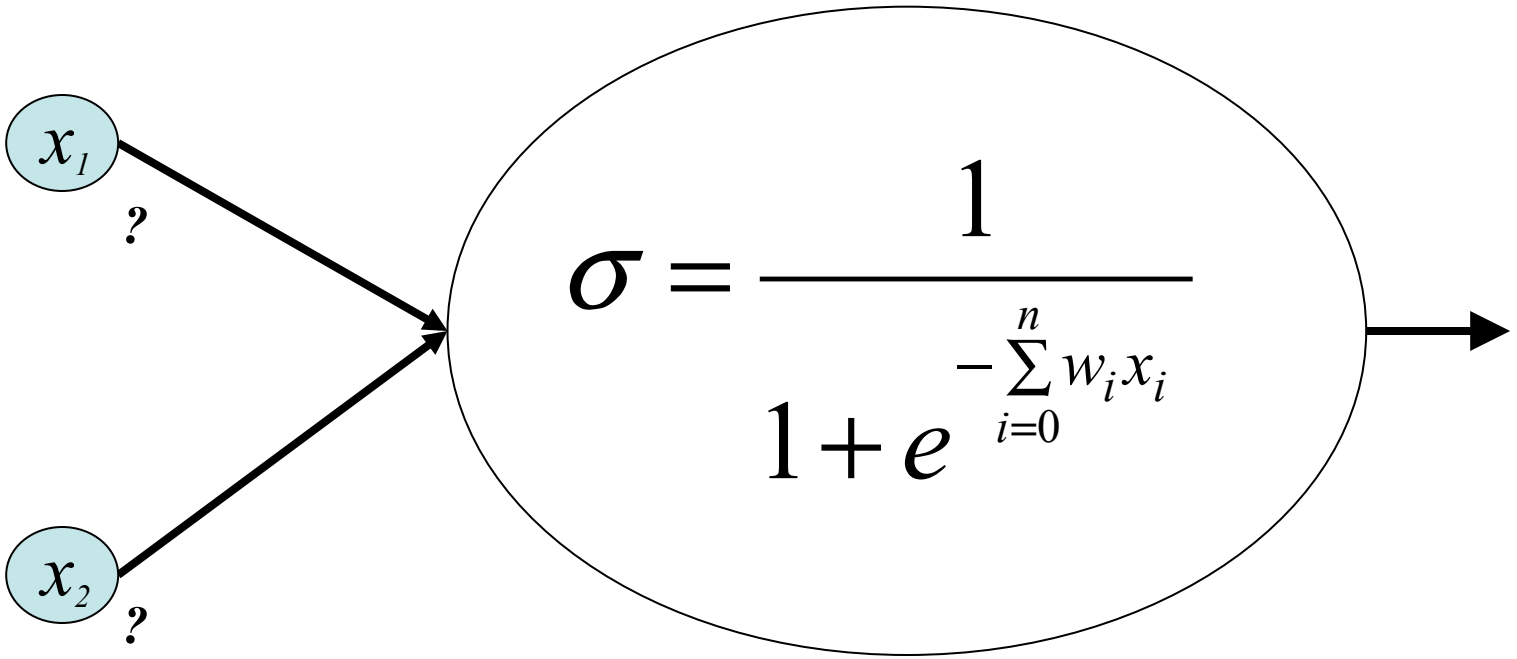
- Sigmoid (Logistic)

$$output = \sigma(net) = \frac{1}{1 + e^{-net}}$$



The sigmoid (logistic) unit

- Has differentiable function
  - Allows assignment of error to predecessors
- Can be used to build non-linear functions
  - Not hand coding of weights!!



Differentiability is key!

- Sigmoid is easy to differentiate

$$\frac{\partial \sigma(y)}{\partial y} = \sigma(y) \cdot (1 - \sigma(y))$$

- …so we basically make an elaborated version of the delta rule to handle learning weights

The Backpropagation Algorithm

For each input training example, $\langle \vec{x}, \vec{t} \rangle$

1. Input instance $\vec{x}$ to the network and compute the output o_u for every unit u in the network
2. For each output unit k, calculate its error term δ_k

$$\delta_k \leftarrow o_k(1-o_k)(t_k - o_k)$$

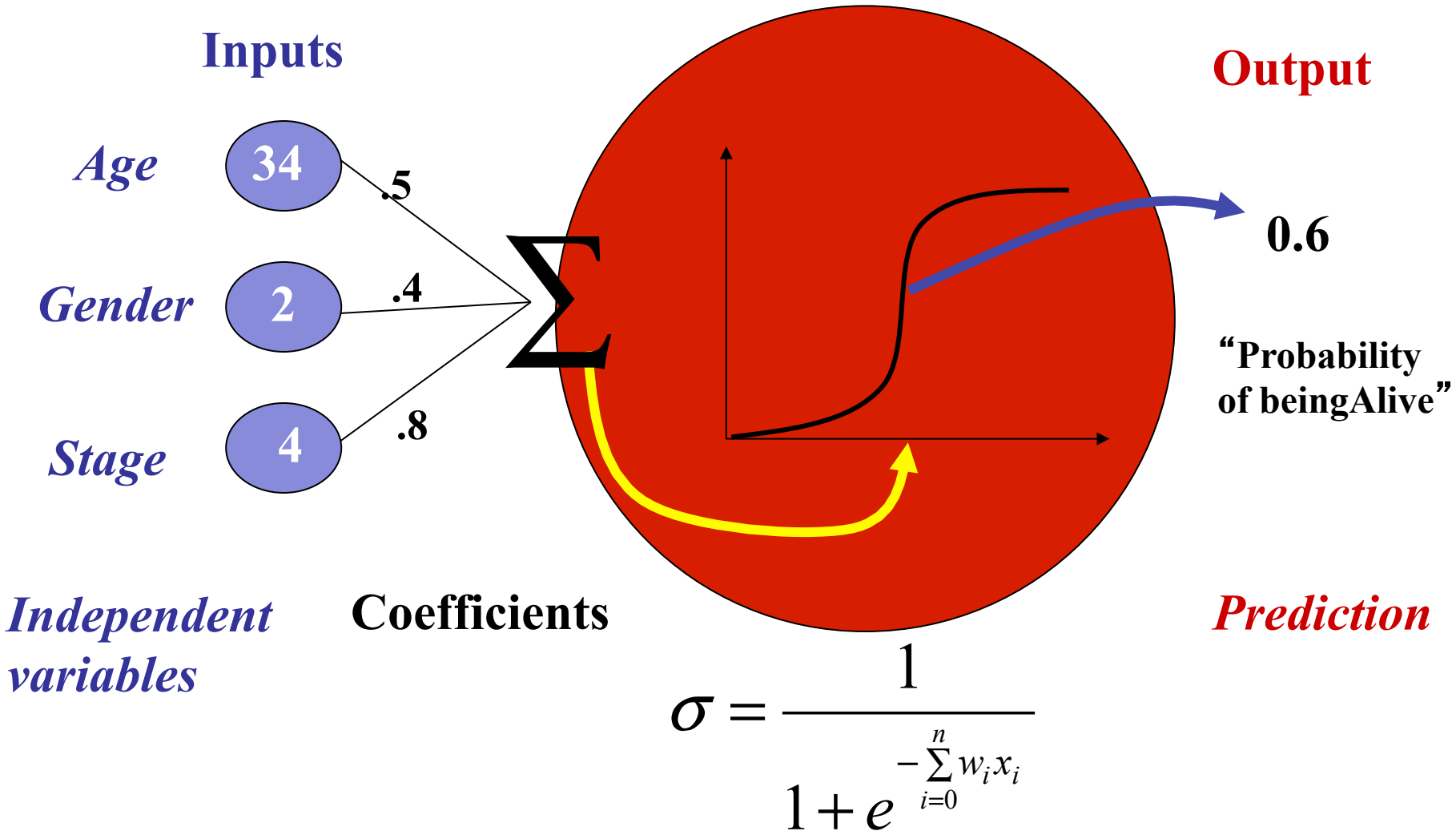
3. For each hidden unit h, calculate its error term δ_h

$$\delta_h \leftarrow o_h(1-o_h) \sum_{k \in outputs} w_{hk}\delta_k$$

4. Update each network weight w_{ji}

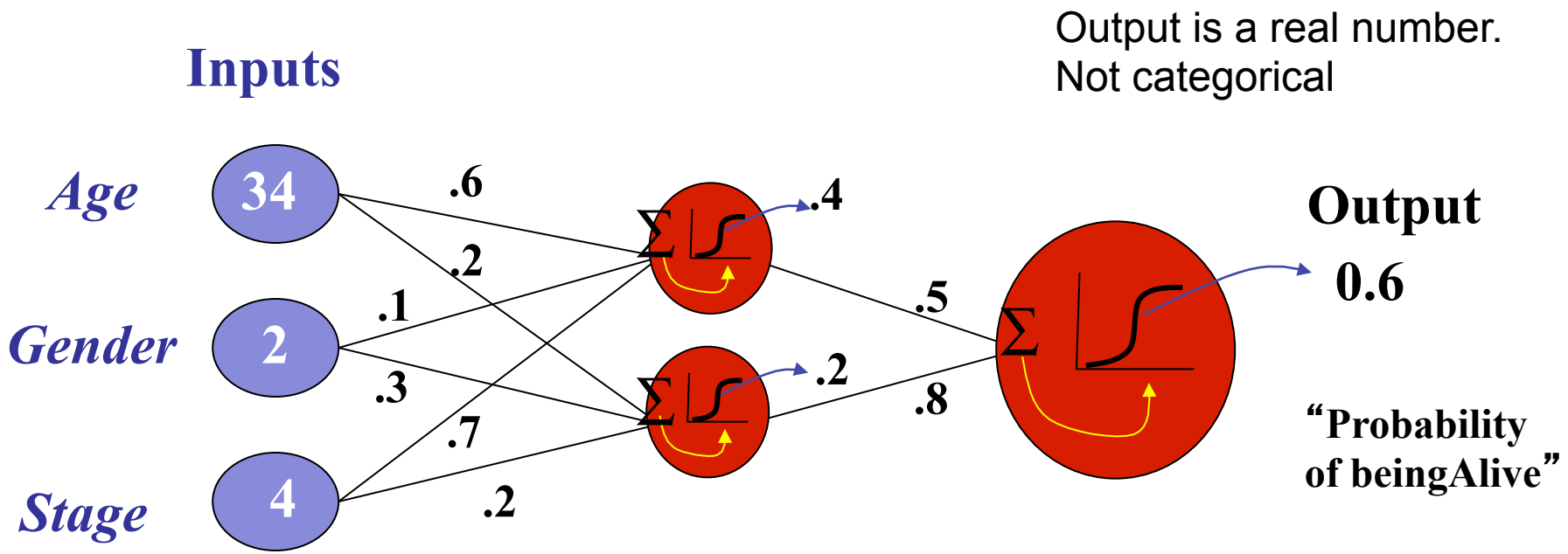
$$w_{ji} \leftarrow w_{ji} + \eta \delta_k x_{ji}$$

Logistic function



$$\sigma = \frac{1}{1 + e^{-\sum_{i=0}^{n} w_i x_i}}$$

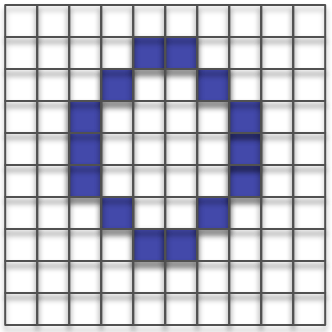
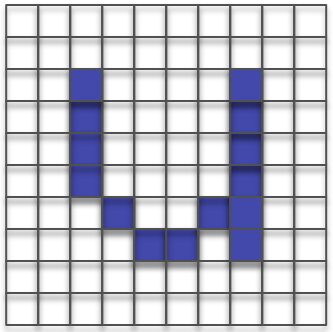
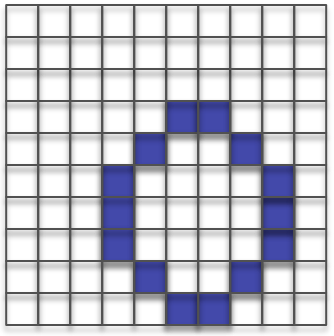
A very simple MLP



How do you decide?

- How to encode the data?
- How many hidden nodes?
- How many hidden layers?
  (just one, or MAYBE 2 for MLPs)
- How big should each layer be?
- How to encode/interpret the output?

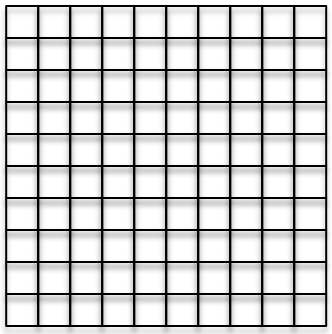
Who is most similar?

- Letters encoded as 10 by 10 pixel grids
- What if they are input to a single perceptron unit?
- Would a multilayer perceptron do better?
- What pitfalls would you envision?

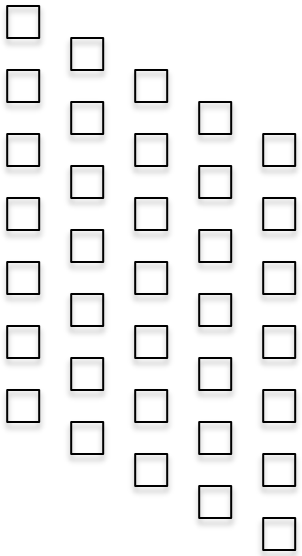
One possibility

INPUT LAYER



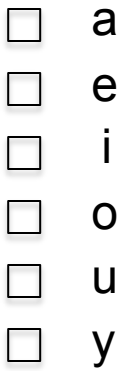
One input per pixel

HIDDEN LAYER



One hidden node per potential shifted letter image

OUTPUT LAYER



One output node per letter

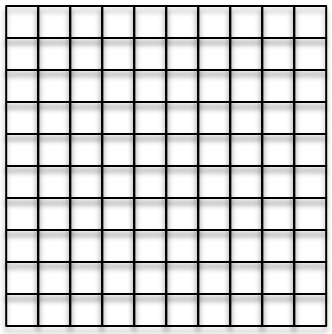
Each node is connected to EVERY node in the prior layer
(it is just too many lines to draw)

Rote memorization

- Is the previous architecture different from rote memorization of cases?
- Will it generalize?
- What is the relationship between the size of the hidden layer and generalization?

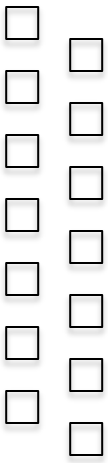
Another possibility

INPUT LAYER



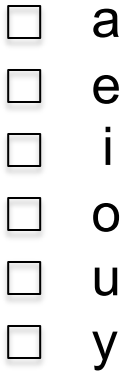
One input
per pixel

HIDDEN LAYER



small number of nodes:
1 per important feature

OUTPUT LAYER



One output node
per letter

Each node is connected to EVERY node in the prior layer
(it is just too many lines to draw)

Interpreting the output

- Treat it as a confidence/probability?
- Force it to 0 or 1 to be a decision?
- Quantize to a range of values?
- How about different output architectures?

Multi-Layer Perceptrons are

- Models that resemble nonlinear regression
- Also useful to model nonlinearly separable spaces
- Can learn arbitrary Boolean functions
- Use gradient descent and are thus susceptible to being stuck in local minima
- Opaque (internal representations are not interpretable)

HEBBIAN LEARNING

Donald Hebb

- A cognitive psychologist active mid 20th century
- Influential book: The Organization of Behavior (1949)
- Hebb's postulate

"Let us assume that the persistence or repetition of a reverberatory activity (or "trace") tends to induce lasting cellular changes that add to its stability.... When an axon of cell *A* is near enough to excite a cell *B* and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that *A*'s efficiency, as one of the cells firing *B*, is increased.

Pithy version of Hebb's postulate

Cells that fire together, wire together.

HOPFIELD NETWORKS

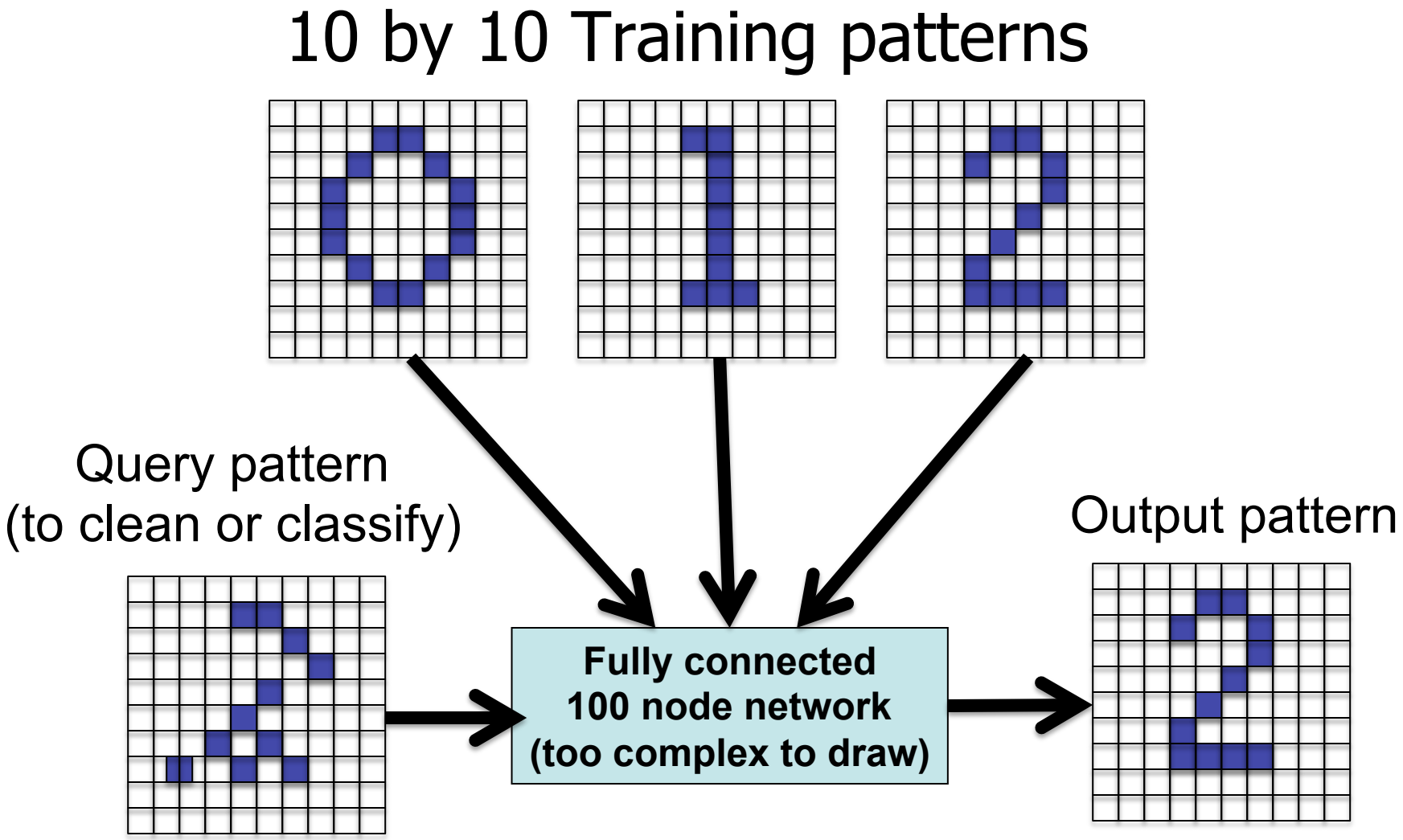
Hopfield nets are

- "Hebbian" in their learning approach
- Fast to train
- Slower to use
- Weights are symmetric
- All nodes are input & output nodes
- Use binary (1, -1) inputs and output
- Used as

  Associative memory (image cleanup)

  Classifier

Using a Hopfield Net



Training a Hopfield Net

- Assign connection weights as follows

$$w_{ij} = \begin{cases} \sum_{c=1}^{C} x_{c,i} x_{c,j} & \text{if } i \neq j \\ 0 & \text{if } i = j \end{cases}$$

c index number for C many class exemplars

i, j index numbers for nodes

$w_{i,j}$ connection weight from node i to node j

$x_{c,i} \in \{+1, -1\}$ element i of the exemplar for class c

Using a Hopfield Net

Force output to match an unknown input pattern

$$s_i(0) = x_i \qquad \forall i$$

Here, $s_i(t)$ is the state of node i at time t

and x_i is the value of the input pattern at node i

Iterate the following function until convergence

$$s_i(t+1) = sign\left(\sum_{j=1}^{N} w_{ij} s_i(t)\right)$$

Note: this means you have to pick an order for updating nodes. People often update all the nodes in a random order, then repeat with a new random order.

Using a Hopfield Net

Once it has converged…

FOR INPUT CLEANUP: You're done. Look at the final state of the network.

FOR CLASSIFICATION: Compare the final state of the network to each of your input examples. Classify it as the one it matches best.

Input Training Examples

Why isn't 5 in the set of examples?

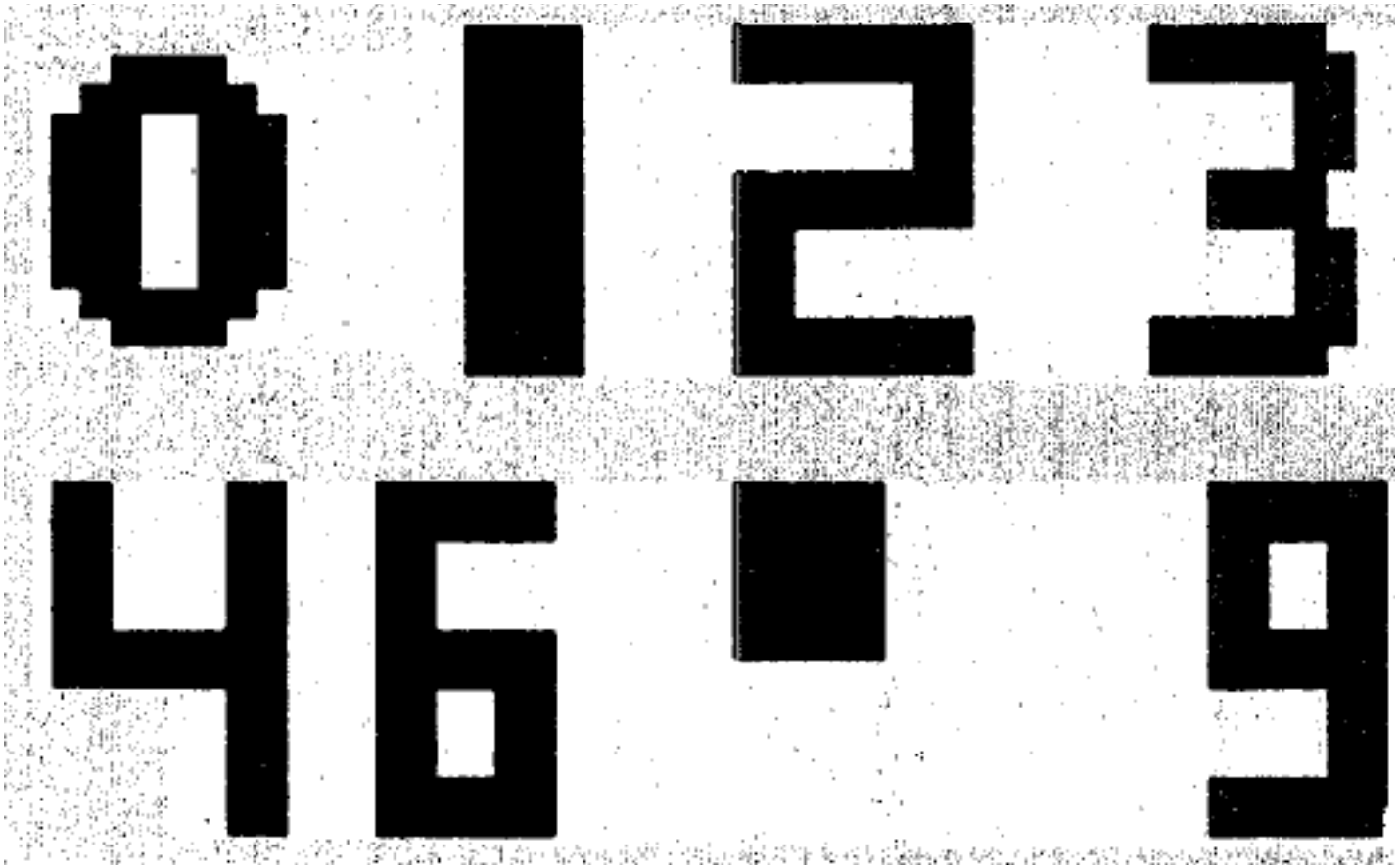


Image from:R. Lippman, An Introduction to Computing with Neural Nets, IEEE ASSP Magazine, April 1987

Output of network over 8 iterations

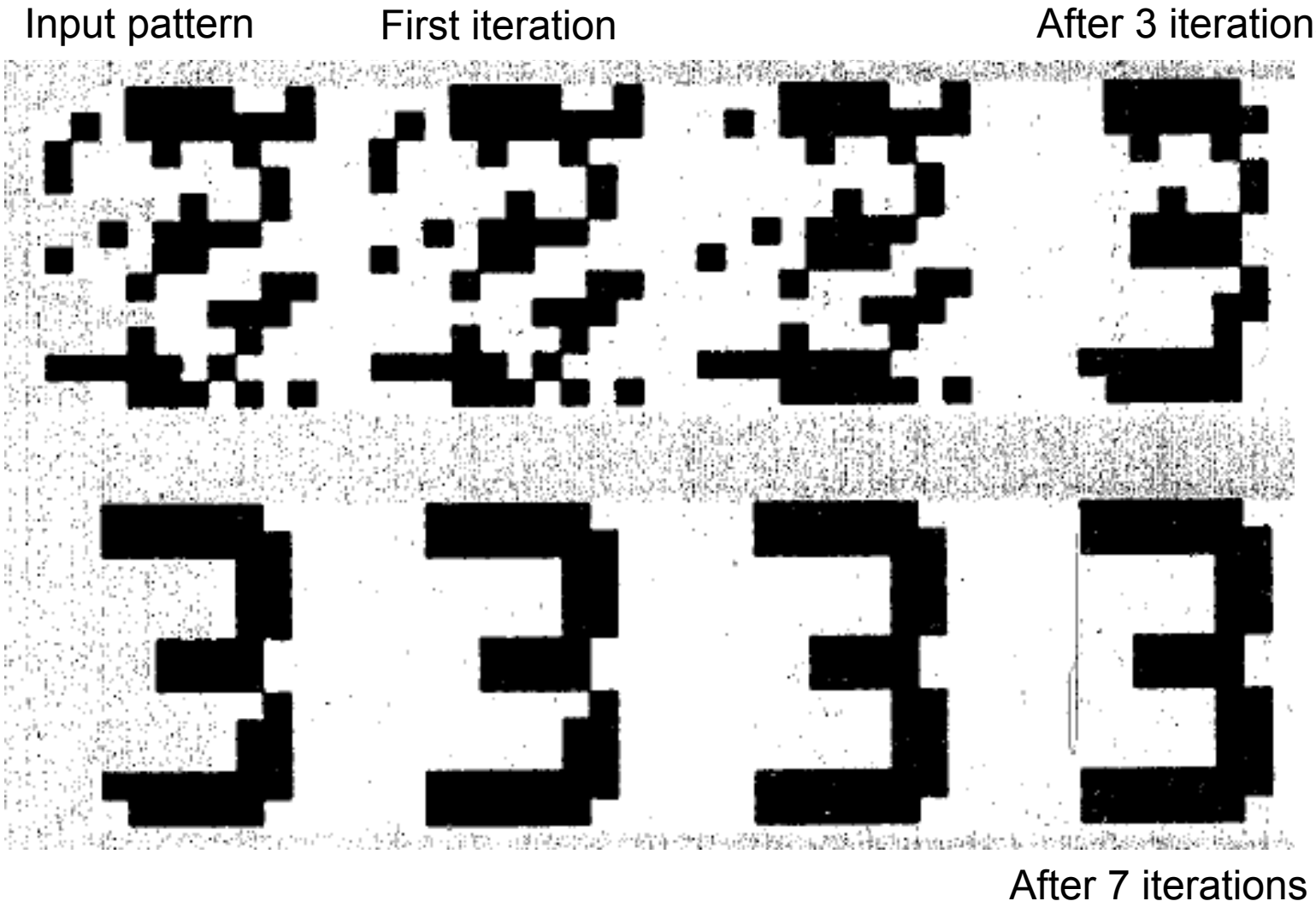
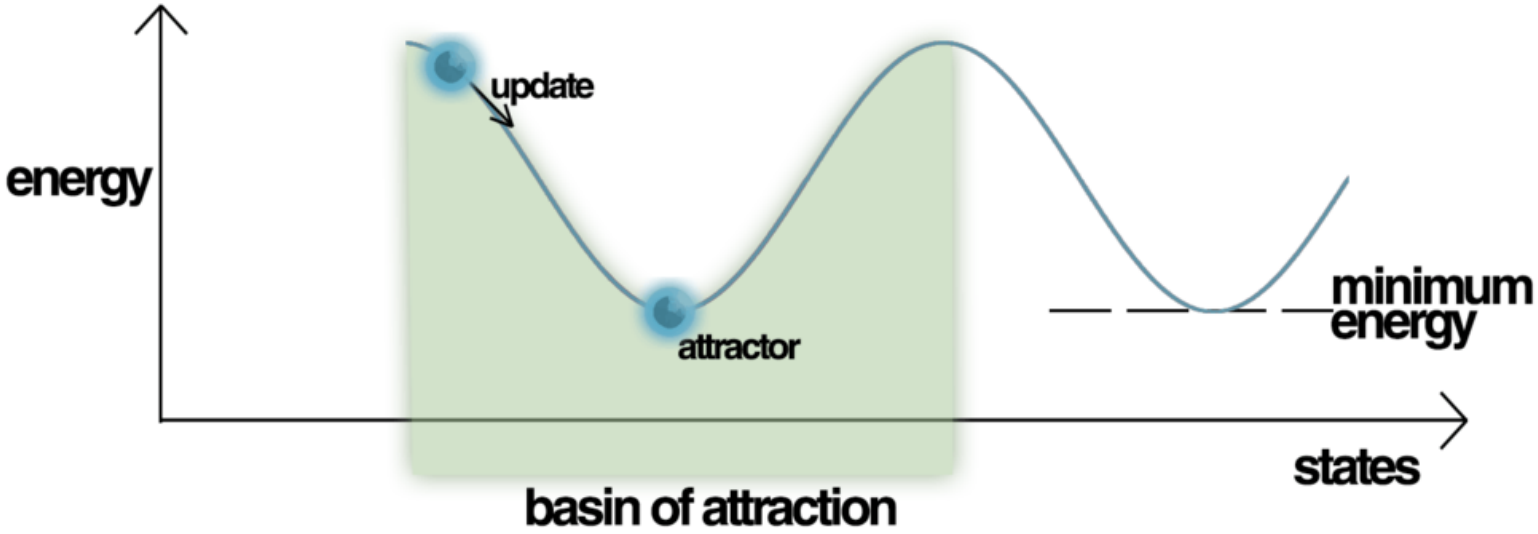


Image from:R. Lippman, An Introduction to Computing with Neural Nets, IEEE ASSP Magazine, April 1987

Characterizing "Energy"



- As $s_i(t)$ is updated, the state of the system converges on an "attractor", where $s_i(t+1) = s_i(t)$
- Convergence is measured with this "Energy" function:

$$E(t) = -\frac{1}{2}\sum_{i,j} w_{ij} s_i(t) s_j(t)$$

Note: people often add a "bias" term to this function. I'm assuming we've added an extra "always on" node to make our "bias"

Image from: http://en.wikipedia.org/wiki/Hopfield_network#mediaviewer/File:Energy_landscape.png

Limits of Hopfield Networks

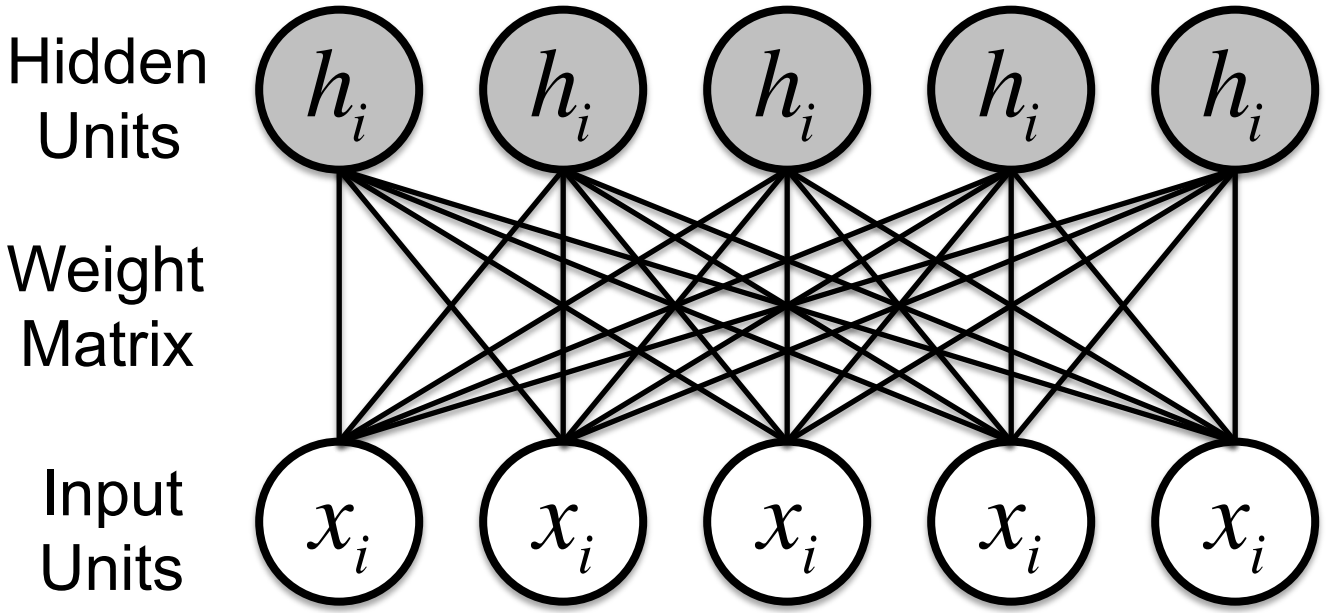
- Input patterns become confused if they overlap
- The number of patterns it can store is about 0.15 times the number of nodes
- Retrieval can be slow, if there are a lot of nodes (it can take thousands of updates to converge)

RESTRICTED BOLTZMAN MACHINE (RBN)

About RBNs

- Related to Hopfield nets
- Used extensively in Deep Belief Networks
- You can't understand DBNs without understanding these

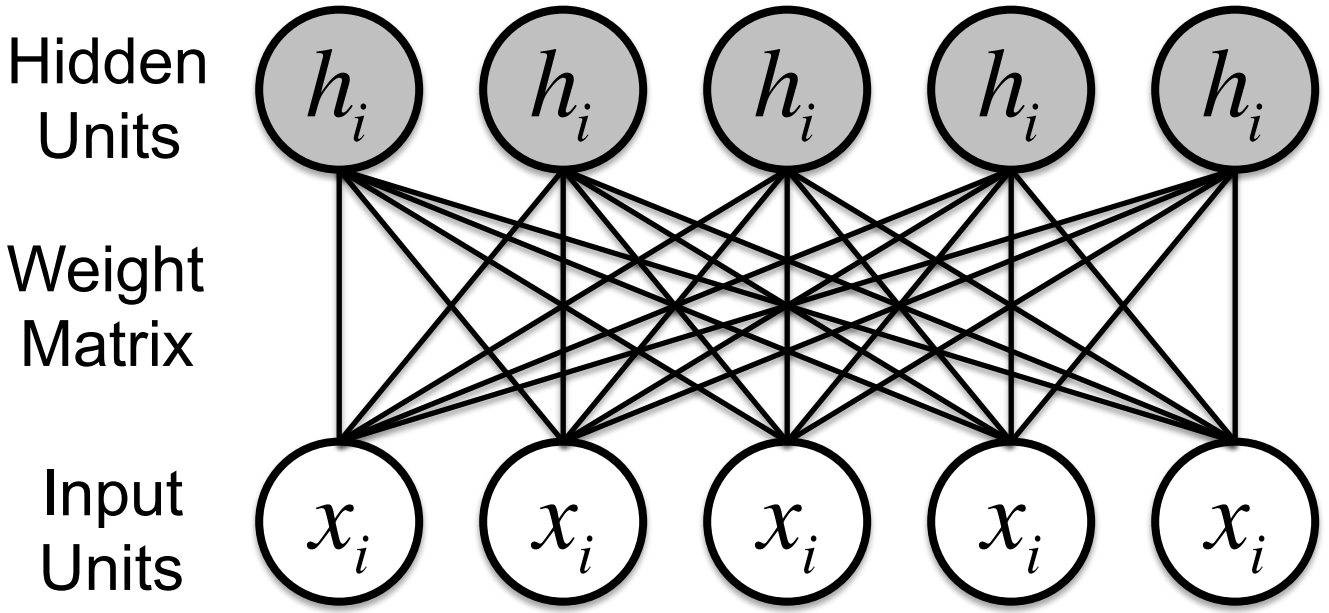
Standard RBM Architecture



2 layers (hidden & input) of Boolean nodes

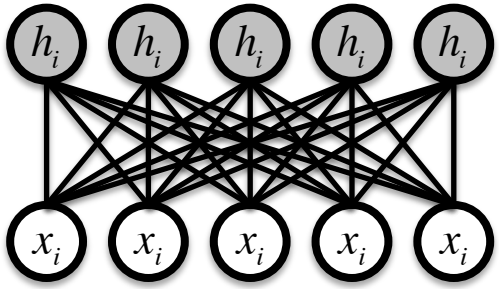
Nodes only connected to the other layer

Standard RBM Architecture



- Setting the hidden nodes to a vector of values updates the visible nodes…and vice versa

Standard RBM Architecture



- Setting the hidden nodes to a vector of values updates the visible nodes…and vice versa

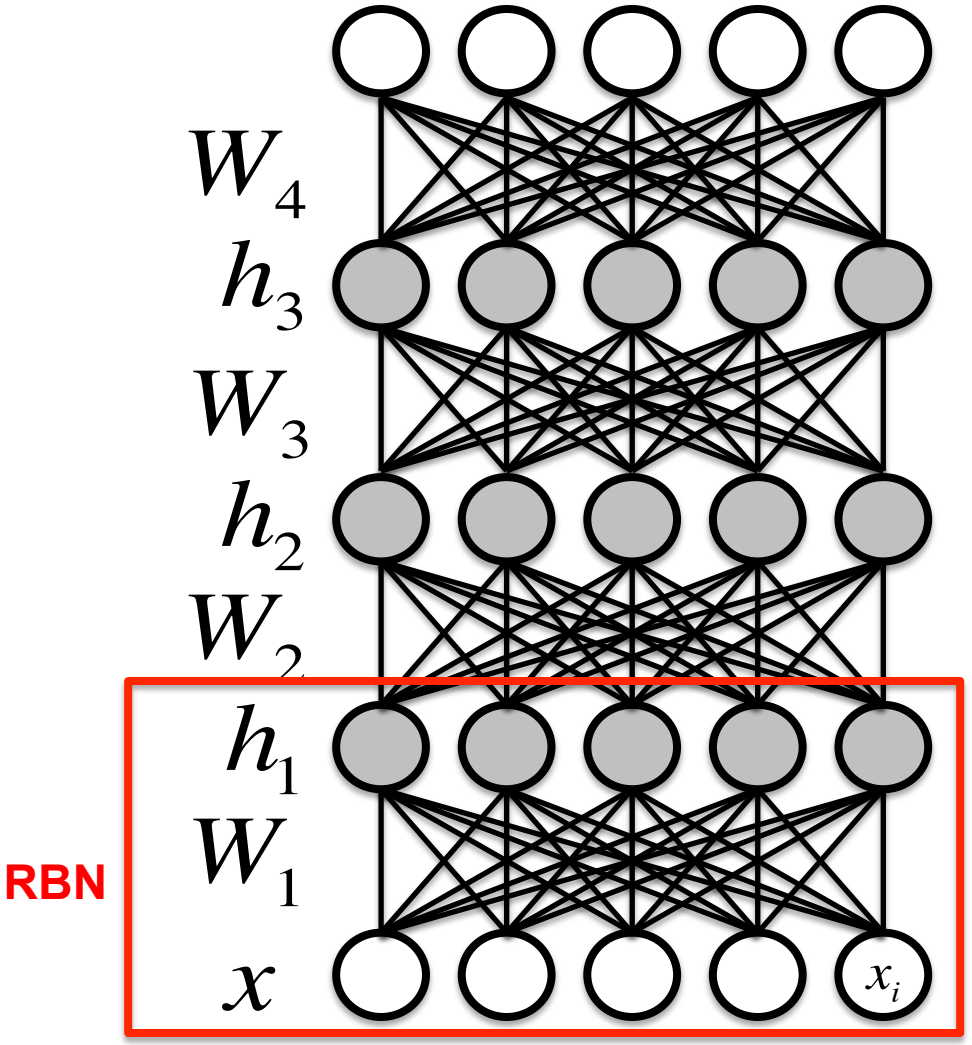
Contrastive Divergence Training

1. Pick a training example.
2. Set the input nodes to the values given by the example.
3. See what activations this gives the hidden nodes.
4. Set the hidden nodes at the values from step 3.
5. Set the input node values, given the hidden nodes
6. Compare the input node values from step 5 to the the input node values from step 2
7. Update the connection weights to decrease the difference found in step 6.
8. If that difference falls below some epsilon, quit. Otherwise, go to step 1.

DEEP BELIEF NETWORK (DBN)

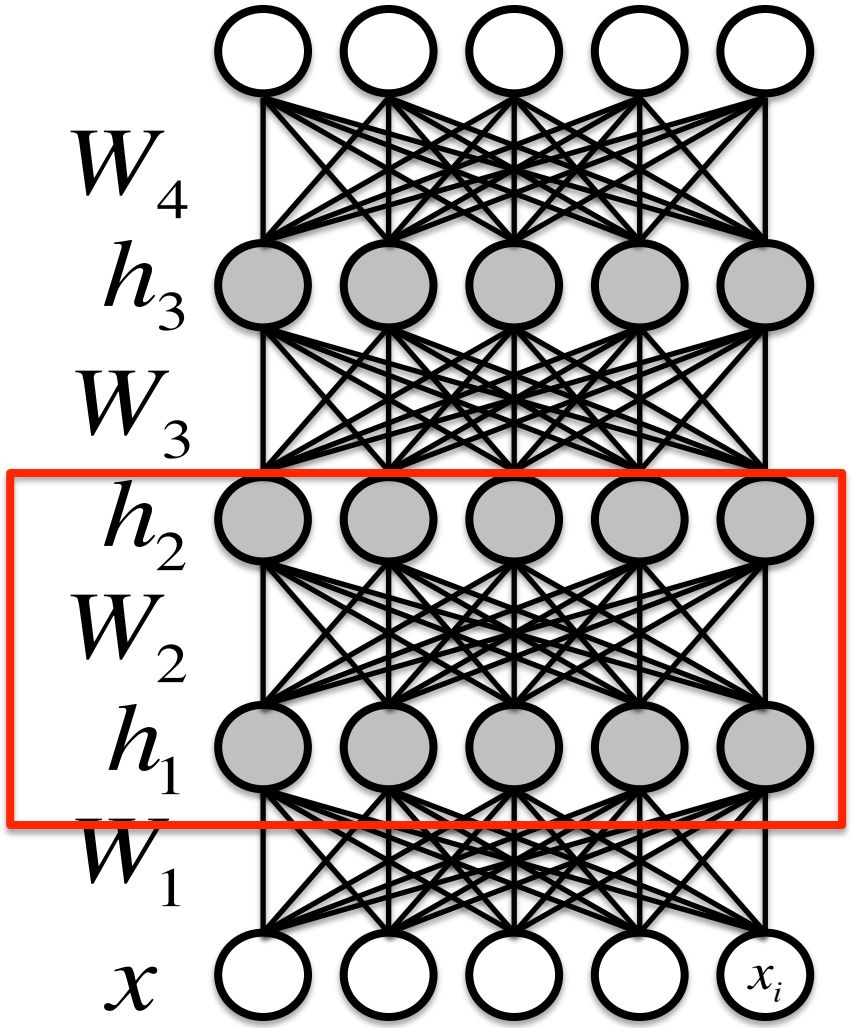
What is a Deep Belief Network?

- A stack of RBNS
- Trained bottom to top with Contrastive Divergence
- Trained AGAIN with supervised training (similar to backprop in MLPs)



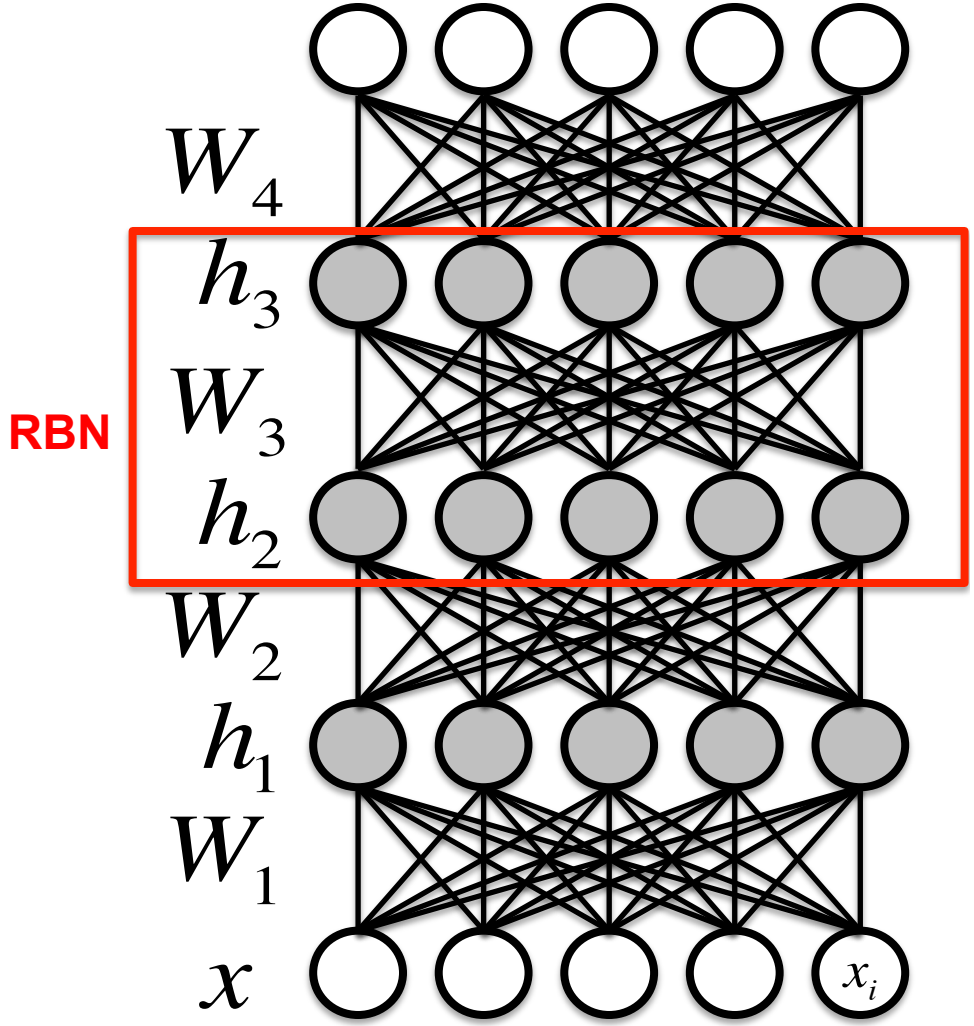
What is a Deep Belief Network?

- A stack of RBNS
- Trained bottom to top with Contrastive Divergence
- Trained AGAIN with supervised training (similar to backprop in MLPs)



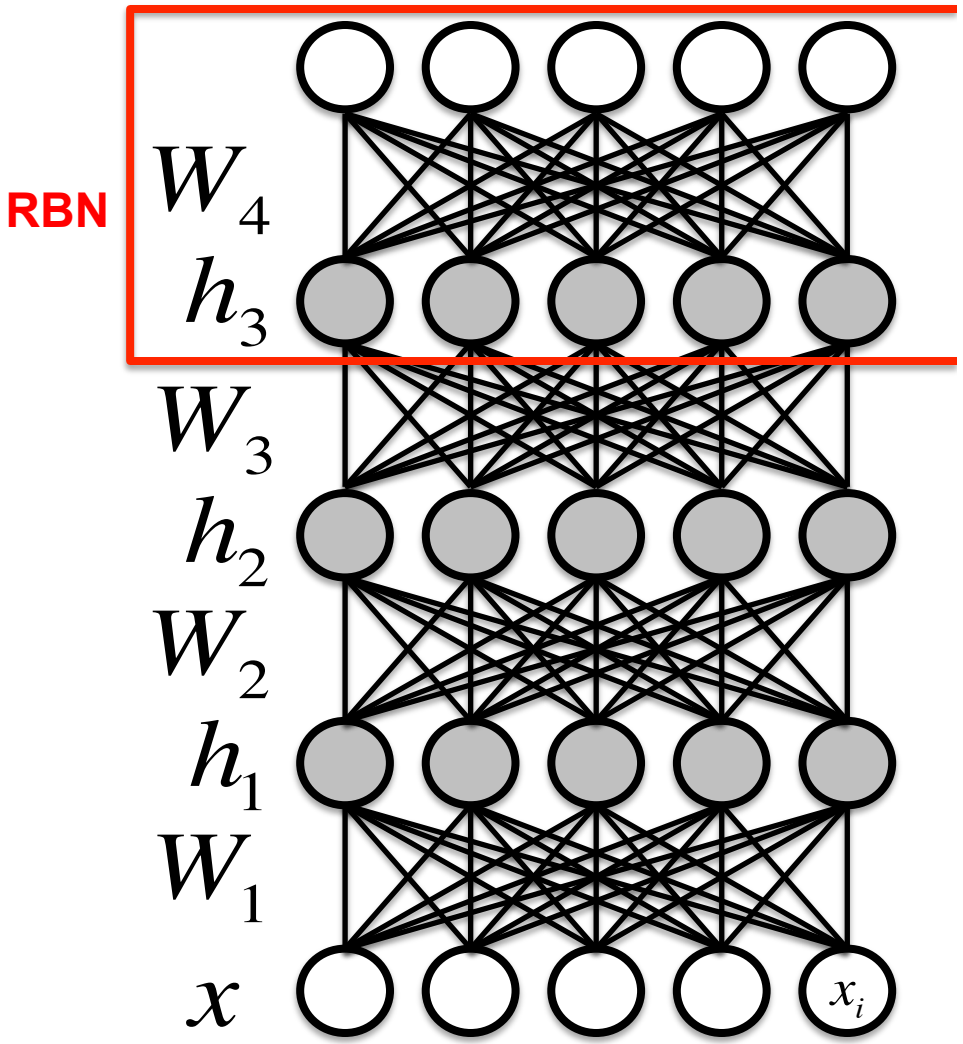
What is a Deep Belief Network?

- A stack of RBNS
- Trained bottom to top with Contrastive Divergence
- Trained AGAIN with supervised training (similar to backprop in MLPs)



What is a Deep Belief Network?

- A stack of RBNS
- Trained bottom to top with Contrastive Divergence
- Trained AGAIN with supervised training (similar to backprop in MLPs)



Why are DBNs important?

- They are state-of-the-art systems for doing certain recognition tasks
  - Handwritten digits
  - Phonemes
- They are very "hot" right now
- They have a good marketing campaign "Deep learning" vs "shallow learning"

How does “deep” help?

- It may be possible to much more naturally encode problems like the parity problem with deep representations than with shallow ones

Why not use standard MLP training?

- Fading signal from backprop
- The more complex the network, the more likely there are local minima
- Memorization issues
- Training set size and time to learn

Benefits

- Seems to allow really deep networks (e.g. 10 layers)
- Creating networks that work better than anything else out there for some problems (e.g. digit recognition, phoneme recognition)
- Lets us again fantasize about how we'll soon have robot butlers with positronic brains.

The promise of many layers

- Each layer learns an abstraction of its input representation (we hope)
- As we go up the layers, representations become increasingly abstract
- The hope is that the intermediate abstractions facilitate learning functions that require non-local connections in the input space (recognizing rotated & translated digits in images, for example)

BIG BIG data

- DBNs are often used in the context of millions of training steps and millions of training examples

RECURRENT NETS

Read up on your own

- Things like multi-layer perceptrons are feed-forward
- Boltzman machines and Hopfield networks are bidirectional
- These have one-way links, but there are loops
- You'll have to read up on your own

SPIKE TRAIN NETS

Read up on your own

- These are attempts to actually model neuron firing patterns
- There is no good learning rule
- There is no time to talk about these

CONVOLUTIONAL NETS

Read up on your own

- Another type of deep network
- References to this work in the course readings